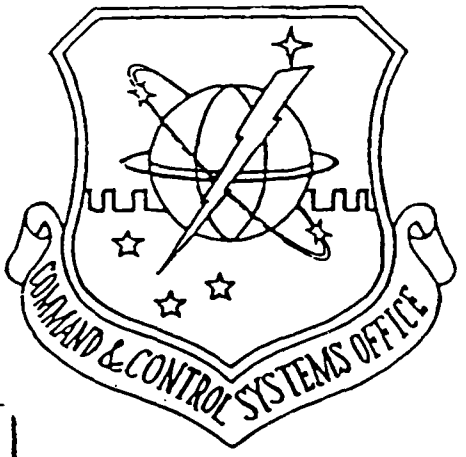
ADA* EVALUATION PROJECT

MODIFIABILITY EXPERIENCES WITH ADA* SOFTWARE

Prepared for

HEADQUARTERS UNITED STATES AIR FORCE
Assistant Chief of Staff of Systems for Command, Control, Communications, and Computers
Technology & Security Division



Prepared by
Standard Automated Remote to AUTODIN Host (SARAH) Branch
COMMAND AND CONTROL SYSTEMS OFFICE (CCSO)
Tinker Air Force Base
Oklahoma City, OK 73145 - 6340
COMMERCIAL (405) 734-2457 / 5152
AUTOVON 884 - 2457 / 5152

* Ada is a registered trademark of the U.S. Government
(Ada Joint Program Office)

29 April 1987

TABLE OF CONTENTS

LIST OF FIGURES

THIS REPORT IS THE EIGHTH OF A SERIES WHICH DOCUMENT THE LESSONS LEARNED IN THE USE OF ADA IN A COMMUNICATIONS ENVIRONMENT.

ABSTRACT

This paper presents some of the techniques used by the Standard Automated Remote to Automatic Digital Network (AUTODIN) Host (SARAH) Development Team to increase the modifiability of their software. The first section of the paper provides some background information on the Ada evaluation task and defines the scope of the paper.

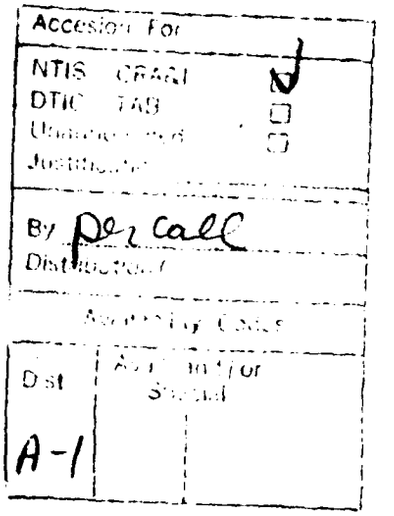
The second section looks at some of the main issues associated with modifiability. A definition of modifiability is established and reasons that software will require modification are given. Some of the benefits of producing modifiable software are covered along with some of the costs and problems.

The third section looks at specific methods that can aid the production of modifiable software. This section states that a sound software engineering methodology is the single most important factor in the development of modifiable software. However, program structure, general coding practices, and internal documentation standards are also important and can significantly add to a software projects modifiability.

The final section summarizes some of the main points and provides some recommendations on the development of modifiable Ada software.

Ada Evaluation Report Series by CCSO

Ada Training	March 13, 1986
Design Issues	May 21, 1986
Security	May 23, 1986
Micro Compilers	December 9, 1986
Ada Environments	December 9, 1986
Transportability	March 19, 1987
Runtime Execution	March 27, 1987
Modifiability	April 20, 1987
Project Management	Spring 87
Module Reuse	Fall 87
Testing	Fall 87
Summary	Fall 87



1. INTRODUCTION

1.1. PURPOSE

The purpose of this paper is to share experiences of the SARAH project concerning software modifiability and the Ada language with other members of the Ada community. This paper is one in a series of papers being written by the Command and Control Systems Office (CCSO) for Headquarters United States Air Force.

1.2. BACKGROUND

Headquarters United States Air Force tasked the CCSO with evaluating the Ada language in the context of real-time digital communications software so that potential Ada developers could gain a practical insight into what is required to successfully develop Ada software. This tasking was divided into twelve evaluation topics, one of which was Modifiability. CCSO chose the Standard Automated Remote to AUTODIN (Automatic Digital Network) Host (SARAH) project as the basis for this evaluation.

SARAH is a small to medium size project (approximately 100,000 lines of source code, and 40,000 lines of executable code) which will function as a standard intelligent terminal for AUTODIN users and will be used to help eliminate punched cards and paper tape as transmit/receive media[1]. The source code for the SARAH project is being written in ADA, except for approximately 600 lines of assembly language code required to drive the communications port. The source code produced is compiled on IBM PC ATs and Zenith 248s using Alsys Ada compilers. The SARAH software will run on a range of PC XT, PC AT, and compatible microcomputers under the MS-DOS operating system (version 2.0 or higher).

1.3. ASSUMPTIONS AND CONSTRAINTS

The assumptions and constraints under which this paper was written are as follows:

1. This paper is based upon experiences gained while developing the first Prototype for the SARAH project. The first prototype does not include the Mode I communications interface or the heavy use of tasking that is required for the Communications version of SARAH.

2. Development has been limited to the Alsys Ada compiler running on the IBM PC/AT and the Zenith Z-248.

3. Observations presented come from different and varied perspectives. The SARAH team members have a variety of previous experience. Some members have had very little software experience. Others are very experienced in system design and development, and have a good working knowledge of different software development environments.

2. SOFTWARE MODIFIABILITY

2.1. DEFINITION OF SOFTWARE MODIFIABILITY

To "modify" means to "change or alter"[2]. Modifiability of software adds to this basic definition a quality of ease with which the change or alteration can be made. Thus, the modifiability of software is the measure of how easy it is to change or alter some feature of the software to achieve a new desired result. At best, software modifiability is a subjective quality. If you make "one little change" and problems ripple through out your program, then you would say that the software showed poor modifiability. If on the other hand, you can quickly locate and change a piece of software and it performs as you expected, you would say that the software was very modifiable.

2.2. REASONS FOR SOFTWARE MODIFIABILITY

There are three basic reasons that software may require modification: a change in program requirements, a change in hardware, or an error in the original program.

As long as software is used by people, there will be requirements for changes in the original design. The DoD may introduce a new message format or change an old one. Operators may want an additional error message or the meaning of a function key changed. The programmer may want a series of debugging routines added. The user may not have known what he wanted in the first place. Whatever the reason, the result is the same, software must be modified to meet new or altered requirements.

Microcomputer technology is advancing rapidly. The modifiability of our software will directly effect whether or not we take advantage of new advances in technology. The SARAH project has already been modified to function on target computers other than those identified in the original SARAH Concept of Operations Document[3]. Initially, the SARAH software was to be targeted only to the Zenith Z-150. To date, the Zenith Z-248 and the Z-200 have been added to this list. The new Z-386, which uses the advanced 80386 microprocessor, will most likely also end up being one of the target machines. The SARAH software has been able to adapt to changes in hardware without rewriting large sections of code or creating a large number of hardware specific versions of SARAH.

Although no programmer likes to admit it, software will also require modification because it never ran correctly in the first place. Modifiability techniques discussed later in this paper will not prevent logic errors, however they will make correction of those errors easier, quicker, and less likely to create additional problems.

2.3. BENEFITS OF SOFTWARE MODIFIABILITY

Significant software cost savings can be realized by producing modifiable software. Studies have shown that 80% of the total system lifecycle costs are directly related to system enhancement and maintenance.[14] If a program is easily modified, maintenance and enhancement costs are lower because it will take less time to modify the current system to achieve the new result and will not cause the problem of having to "fix the fix."

An example of how the SARAH project is using modifiable software to reduce development costs can be seen in the message mask processing of the SARAH workstation. The DD 173 Message Mask, which provides a mask prompting the user for the various fields required to complete a DD 173 form, required approximately 4 man-months to design and code for the first SARAH prototype. Modifying the original design to provide a mask for the DD 1392 took approximately 2 man-months. This trend is expected to continue for several additional masks that must be developed.

2.4. COSTS OF SOFTWARE MODIFIABILITY

The benefits of software modifiability are not free. In order to develop modifiable software, you must be willing to accept larger source and executable files, longer design and development time, and increased training costs.

Both the source files and the executable files of easily modified software are usually larger than their less easily modified counterparts. Modifiable software projects use software engineering principles such as abstraction, information hiding and modularity to achieve their goal of software modifiability. Additional code will be required to implement these principles. Additional code translates into larger executable files. The larger source files cost more to manage and store. Larger executable files may require more memory or a faster system to operate.

Another cost of easily modified software is increased development and training time. A modifiable software project requires planning and coordination to implement the engineering principles that promote software modifiability. The Ada language provides the tools with which solid maintainable software can be constructed. However, use of the Ada language does not guarantee modifiable software. Designers and programmers must be trained to use these tools. This training will take time and experience. "Ada Training for Development Teams"[14] discusses specific training issues and addresses the training experiences of the SARAH Project Team.

3. WAYS TO IMPROVE SOFTWARE MODIFIABILITY

In this section, we will discuss specific actions taken by the SARAH development project to make the code for SARAH more modifiable. Some of these recommended actions are relatively simple while others are more complex. We feel that by following these suggestions we have made our software easier to modify now and easier to maintain and enhance in the future.

3.1. SOUND SOFTWARE ENGINEERING TECHNIQUES

Modifiable software is well-engineered software. Using solid software engineering techniques to design software is the single most important thing that can be done to insure its modifiability. Like a poorly designed house, poorly designed software can be shored up, patched, and poor features hidden. However, like a poorly designed house, poorly designed software, no matter how much it is "fixed", remains poorly designed software that will cause its owner continual problems. The software engineering techniques used in the development of the SARAH project were strongly influenced by the Structured Design Techniques of Yourdon[7] and Object Oriented Design techniques of Booch[8]. More details on the specific software engineering approach used in the SARAH project is discussed in "An Architectural Approach to Developing Ada Software"[3].

3.2. PROGRAM STRUCTURE

3.2.1. DEVELOP MODULAR COMPONENTS

It is doubtful that a single computer programmer can comprehend each detail of today's large complex software systems. Even the rather modest 20,000 lines of executable source code found in the first SARAH prototype would challenge most programmers. This is a problem for software modification because a programmer should not attempt to modify any code that he does not understand or the resulting change may cause a rippling effect that will bring down the entire system. Decomposing a software system into component modules allows us to apply modern programming principles such as abstraction and information hiding to create software that is easier to understand. Breaking a software system into modules allows us to attack a largely incomprehensible problem as we might a jigsaw puzzle. When first dumped out of the box, the jigsaw puzzle picture is rather incomprehensible. Then, piece by piece, the picture takes shape. The programmer that could not comprehend the whole system can study each of its composite modules and then join the pieces to finally understand the whole picture.

Breaking software into component modules promotes modifiability because it limits the scope of the modification. A programmer may not need to fully understand the entire system in order to

modify a module of the system. Well defined modules with explicit interfaces can be coded and debugged as independent units. Coding errors should be detected at the module level and corrected before integrating the module into the overall system.

3.2.2. ISOLATE ALL MACHINE DEPENDENT CODE

Since SARAH is being developed for IBM PC compatible machines, some of the code must address IBM specific machine and operating system dependencies, such as the address of memory mapped video for the Mono and CGA adapter cards. In order to construct a more easily modifiable system, the SARAH designers identified a logical kernel to isolate the application from the machine and DOS dependencies. When making machine specific modifications, such as adding an Enhanced Graphics Adapter capability, only the code in the kernel packages needs to be changed.

3.2.3. DEFINE COMMON TOOLS

Project designers should always be on the lookout for common code. Any code that provides similar functions or data objects for two or more packages should be grouped into a common tools package. By isolating common code in one place the location of any modification to the functions performed by the tools is readily found and changed. If on the other hand, we did not have common tools packages and code performing the same functions was spread throughout the system, any change would greatly increase the time required to make a modification because each place the code is located must be found and changed. The chances of making an error is increased with each change.

An example of a common tools package identified by the SARAH project designers is the Message_Validation package. A message must be validated before the Utilities package will print it, fields of a message mask must be validated before the Edit package will save it, and the security of an incoming message must be checked against the maximum allowable security for the terminal before the COMM package will accept it. These and other message content checks require validation of the same fields of a message. By placing the tools to perform the validation of various message fields in a common package we can control message validation. If DoD defines a new message precedence of Q, the only code that needs to be modified is isolated in the Message_Validation package.

Combining similar functions into a common routine or package is a technique which should not be limited to the system designers. Programmers can enhance the modifiability of packages and subprograms that they code by isolating similar code. In the SARAH message mask processing, three procedures need to know the starting row and column in the Text Window where data for a given field should be displayed. The three procedures place data into the data area of the Text Window at various times, so it is

imperative that they all start in the same place. A common procedure, Calculate_Starting_Positions, was defined to insure that each of the using procedures displays data in the proper position on the screen.

3.3. GENERAL CODING PRACTICES

3.3.1. SAY WHAT YOU MEAN

Definitions of Ada types should include range constraints to prevent machine dependencies from being inadvertently written into a program. To illustrate, consider the following example. If the compiler manufacturer implements the pre-defined type Integer based on word length, the loop in the example would be executed 32767 times on a 16 bit machine, but only 127 times on an eight bit machine.

```
subtype Loop_Counter is Integer;
for  i in  1..Loop_Counter'last loop

  . . . Execute some code

end loop;
```

By constraining the Loop_Counter, two things are accomplished. First, the code is not dependent upon the compiler's implementation of predefined types. Second, the execution of the code is made less ambiguous. For example:

```
subtype Loop_Counter is Integer range 1..127;
for  i in  1..Loop_Counter last loop

  . . . Execute some code

end loop;
```

3.3.2. AVOID THE USE CLAUSE

The "Use" clause allows us to achieve direct visibility of components specified in compilation units (packages and subprograms) that we have "withed"[10]. Even though this simplifies referencing components by allowing the use of a component's simple name, direct visibility can cause problems when trying to determine the source of an imported subprogram, type, or object identified only by its simple name. For example, the SARAH View_Operations package that contains routines which allow the operator to view files and directories lists ten packages in its context clause. If these packages were also listed in a Use Clause at this point, how would we know where "low" came from in this instruction?

```
Set_Int_Attrib_Transient(Window_ID,low);
```

It would be very difficult without actually searching through the ten packages in the Context Clause. Generally, the Use Clause is avoided by SARAH programmers in order to clarify the source of each component.

While explicitly naming each component of the previous instruction makes their source clear, it does little for overall readability:

```
Transient_Window_Manager.Set_Int_Attrib_Transient(Window_ID,
  SARAH_VDT_Constants.low);
```

Restricting the use of the Use Clause does not mean that source code must be cluttered with long extended names for every component imported into a package. Renaming components can improve readability of Ada source code while maintaining easily identified components. Applying this technique to the View_Operations package, we "withed" the ten packages we wished direct visibility to. Then, at the top of the package we renamed the packages using their acronym. Using a packages acronym in the extended name gives a crisp short name that can quickly be traced back to its parent. For Example:

```
with Utilities_Disk_IO, Transient_Window_Mgr, Transient_Msg_Mgr,
     Prompt_Window_Mgr, Help_Window_Mgr,      Menu_Mgr,
     Key_Mgr,           SARAH_VDT_Constants,  Sys_Bufr,
     Out_Of_Memory_Tools;


package body View_Operations is

                          ----------------
------------------------- Rename Packages ---------------------
                          ----------------
                          ----------------------------------------
                          -- packages are renamed to reduce
                          -- clutter, but are not "used" because
                          -- there are so many external routines
                          -- called, it would be confusing which
                          -- came from which package
                          ----------------------------------------
  package UDIO renames Utilities_Disk_IO;
  package TWM  renames Transient_Window_Mgr;
  package TMM  renames Transient_Msg_Mgr;
  package PWM  renames Prompt_Window_Mgr;
  package HWM  renames Help_Window_Mgr;
  package MM   renames Menu_Mgr;
  package KM   renames Key_Mgr;
  package SVC  renames SARAH_VDT_Constants;
  package SB   renames Sys_Bufr;
  package OOMT renames Out_Of_Memory_Tools;

                          .
                          .
```

```
                    .

                    -- give buffers back to the system
    SB.Return_List(Directory);

                    -- close the transient window and reset the
                    -- transient window intensity to low
    TWM.Close_Transient_Window(Window_ID);
    TWM.Set_Int_Attrib_Transient(Window_ID,SVC.low);


exception
  when UDIO.Operator_Abort => null;
  when UDIO.Buf_Failure => RAISE SB.Buf_Failure;

                    .
                    .
                    .
```

3.3.3. INITIALIZE ALL VARIABLES

All objects used in an Ada program should be initialized with a value when they are declared, unless you can be absolutely certain that they will be initialized before use. The Ada Reference Manual does not specify a default value for Ada objects. Since this is left up to the various compiler manufacturers, the value of uninitialized objects is undefined. The unreliable results caused by uninitialized variables may be compounded in large systems where procedures nested within another procedure or package rely on variables in still another package.

3.4. DOCUMENTATION

Documentation is another important means of insuring software modifiability. Documentation of software can be divided into two types, external documentation and internal documentation. External documentation consists of the plans and documents associated with the software that are not part of the actual code. A problem with External Documentation is its tendency to become outdated when changes are made in the code and not in the External Documentation. This problem is compounded when External Documentation is manually maintained. External Documentation for the SARAH project was written using the guidance of Mil-Std 2167 and will not be directly addressed by this paper.

Internal Documentation consists of any of the comments or coding standards listed in the actual code. The following paragraphs discuss some of the Internal Documentation actions the SARAH project has taken to increase the quality of modifiability in our software.

3.4.1. STANDARDIZE DOCUMENTATION

Documentation and coding standards should be established and followed by all members of the development team. Common coding standards lend an air of familiarity to code that makes the entire software system easier to understand, debug, and maintain. It is important to insure that the entire project team understands and agrees to follow the projects documentation standards because the success or failure of documentation standards depends upon the attitude of the individual designers and programmers.

The Draft Ada Design Coding Standards outlined in Appendix D of Mil-STD-2167[6] was not available to guide us in our effort to establish a coding standard for the SARAH project. We started with the Intellimac Style Guide[5] and added our own ideas. Many of our added features are similar to those ultimately listed in the proposed Mil-Std Appendix. Several draft documentation standards were reviewed by the project staff before a final standard was established. This expanded the base of experience upon which the standards were made and increased the acceptance of the final standard. Some of the more salient features of the documentation standard used in the SARAH Project are discussed in the following paragraphs.

3.4.2. USE COMMENTS TO DEFINE STRUCTURE

One of the first standards we established was the overall documentation structure that our code would follow. We defined structures for packages, subprograms, major sections, and comments found within our SARAH code. It is not important that an Ada software project follow any given standard, only that the project follow a reasonable standard that will ease the identification of various parts of each software element. A draft of the standard used by the SARAH project, including an example of its use, is attached as Appendix A.

While not a required standard, we have found that including a diagram based upon Burkhardt and Lee Ada structure charts[11] in the description box of our package specifications greatly enhanced their usability. The structure chart for the COMM_Support package is depicted in figure 3-1. The Initialize, Start, and Shutdown procedures drawn partially in and partially out of the package are visible to entities that "with" COMM_Support. The right-hand portion of the package represents the package body. The Message_Distribution package and the Statistics_Report procedure are totally within the body of COMM_Support and are not visible outside that package. We organize the packages and procedures inside of the package specification and body in the order shown in the structure chart. The COMM_Support package would list the Message_Distribution package, followed by the Statistics_Report procedure, followed by the Initialize, Start and Shutdown procedures. Thus, the Burkhardt and Lee charts act as a quick pictorial reference to the contents of the package.

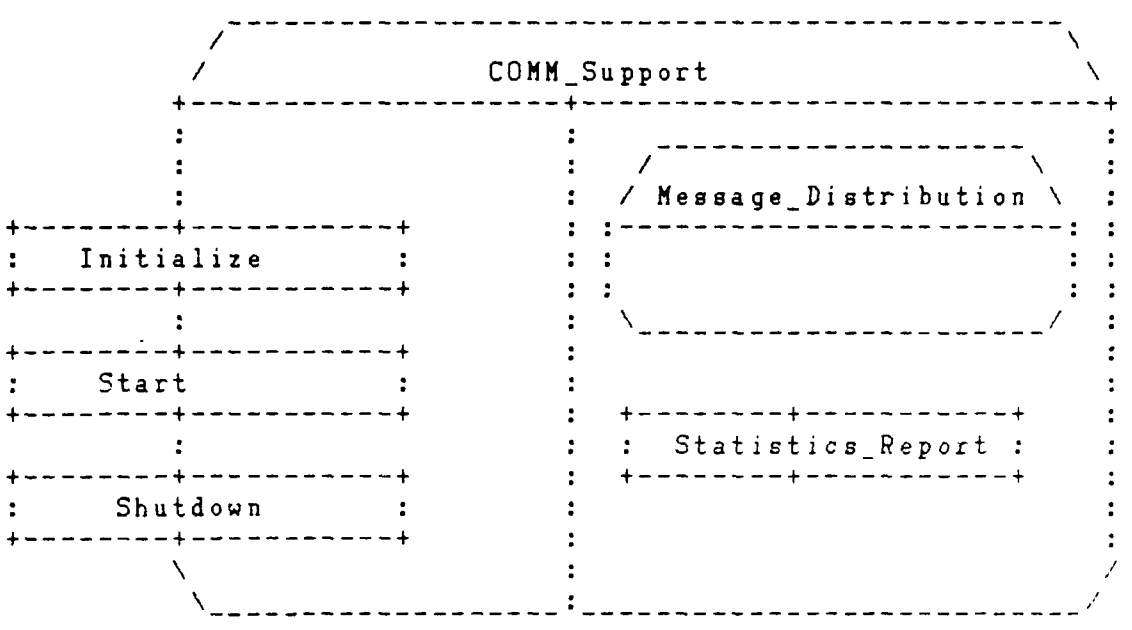


Fig. 3-1: Burkhardt and Lee Ada Structure Chart

3.4.3. USE COMMENTS TO EXPLAIN CODE WITHIN MODULES

Ideally source code and comments complement each other. One should gain a certain amount of information from "reading" code. Comments should provide an additional perspective. Together, source code and comments should present a comprehensive picture of the functions taking place. The comment in the following example is almost useless because it simply repeats the code it is describing and adds nothing to the overall understanding of the code segment.

```
-- set Apples to 0
Apples := 0;
```

A more meaningful comment might appear as:

```
-- initialize number of fruit for salad
Apples := 0;          -- out of season
```

The SARAH project team has found the time required to develop worthwhile comments to be well worth the effort. Because of limited programming resources and a tight time schedule, the programmer that starts coding a module may not be the same programmer that finishes or modifies it. The increased understanding provided by meaningful comments reduces the the time required for modifications and enhancements by smoothing the transition between programmers.

The standards developed for commenting SARAH source code allow a certain amount of flexibility. Since this is the first project using the Ada language developed by CCSO, we were not certain exactly what form we really wanted comments to take and hesitated to lock ourselves into a rigid format that would be difficult to follow, such as requiring a comment for every line or source code.

A convention that comments will start in, or after, column 25 has worked out well. By restricting comments to the right side of the page, a programmer can concentrate on the source code if he is working on the coding details or read through the comments if he simply wants an overview.

A commenting style adopted by many of the SARAH programmers uses a comment block to document 3-8 lines of source code instead of commenting each line individually. This style has proven to be a good balance between describing source code in detail and time required to prepare comments. For example:

Example of Block Comments to Explain Code:

```
                        ------------------------------------
                        -- draw line for input field
                        ------------------------------------
    Column := Input_Column;
    for i in 1..Form_Table(Screen(Display_Element)).input_length  loop
      Put_CharXY_Text(Column,Input_Row,Screen_Space);
      Column := Column + 1;
    end loop;
```

Example of Commenting Each Line of the Above Code:

```
    Column := Input_Column;  --  start in column 1
    for i in 1..Form_Table(Screen(Display_Element)).input_length  loop
                             --  do the following for each  space
                             --  in the current input field
      Put_CharXY_Text(Column,Input_Row,Screen_Space);
                             --  put in an underline
      Column := Column + 1;  --  point to next column in input field
    end loop;                --  end of loop to draw a line
```

3.4.4. USE COMMENTS TO DOCUMENT RATIONALE

The SARAH Software Development File contains notes detailing the rationale for some of our coding decisions. However, we also include some of this information in our code. This helps to prevent misunderstandings caused by the misplaced or outdated documentation inherent with a manual documentation system. Often formal documentation of a module is maintained in a separate location from the program source code for the module. Thus, documenting the rationale for even "slightly" obscure programming

techniques will help programmers quickly understand why code was written as it was. For example, the following comments are extracted from the SARAH VDT_Manager and explain how we can determine whether a given video adapter is installed. Without the comments, it might appear strange to see code that writes a character to memory and then immediately reads a character from that very same spot in memory. Why would the character ever be different?

```
-----------------------------------------
-- The video memory addresses used by the
-- IBM PC and compatible machines are
-- logical addresses (do not physically
-- exist in the machine memory). The
-- physical video memory is on the video
-- adapter card. Using this fact, we can
-- check to see if the video adapter card is
-- installed and working by storing
-- something to a memory location on the
-- video card and then reading the location
-- to see if we get the same thing back. If
-- the video adapter card is not installed,
-- anything we write goes in the Ol' Bit Bucket.
-----------------------------------------
```

3.4.5. USE DESCRIPTIVE NAMES

Descriptive identifiers can substantially increase the readability, understandability and therefore the modifiability of a program. As pointed out in Ada in Practice, "Good identifiers serve as comments, making programs largely self-explanatory". This is particularly important in Ada where context clauses allow the use of identifiers declared in any number of separately compiled units.

Ada gives us the freedom to use meaningful names of any length (up to one line) for all Ada entities, including types, objects, subprograms, packages, and tasks. Often this capability is not effectively used because of long established coding habits. The backgrounds of the majority of the SARAH programmers is deeply rooted in assembly language, which forces the use of short, cryptic names. Thus, there was a period of learning and experimentation at the beginning of the coding phase where names such as Ds1 finally gave way to more descriptive names like Dynamic_String_1.

Care must be taken however, to not create names longer than needed to convey the meaning desired. Excessively long identifiers contribute as much to code unreadability as short cryptic identifiers do.

```
It_Is_Not_Time_To_Quit_Editing_The_Current_Multi_Line_Field_Yet
```

may be a very descriptive loop control object. However, it is awkward to use because it barely fits on a line and takes a relatively long time to type. Editing_Multi_Line_Field is an alternative that, while still descriptive, can be used more effectively:

```
      while Editing_Multi_Line_Field loop
```

Descriptive names can also help to identify the source or purpose of the item named. On the SARAH project, we have found it useful to specify the relationship of closely related entities with a suffix that indicates their function:

xxxx_type	= name of a type or subtype
xxxx_ptr or xxxx_pointer	= name of access type pointing to something of xxxx type
xxxx or something_xxxx	= object of xxxx_type.

Putting these suffixes to use in the following example, you can see how easy it is to discern that My_Buffer is an actual buffer while My_Buffer_Ptr and Some_Other_Buffer_Pointer are pointers to buffers.

```
type Buffer_Type is  . . .
type Buffer_Ptr_Type is access Buffer_Type;

My_Buffer                 : Buffer_Type;
My_Buffer_Ptr             : Buffer_Ptr_Type;
Some_Other_Buffer_Pointer : Buffer_Ptr_Type;
```

We have found this descriptive naming convention particularly useful for types and objects that will be used by several packages. The Buffer_Manager is a common tools package that manages linked lists of buffers for the entire SARAH system. A common system buffer and a pointer to the buffer is defined by the Buffer_Manager package. Anywhere in the SARAH system, we can easily determine whether we are working with a buffer or an access type pointing to a buffer simply by looking at an objects name.

3.4.6. USE SPACE TO YOUR ADVANTAGE

In addition to specific documentation structures mentioned above, spacing can be used effectively to enhance the readability of Ada code at all levels. Source code that is easy to read and understand is easier to modify. Effective use of spacing may consist of skipping three lines between subprograms or indenting if, loop, and block statements to show structure. Spacing can also be used to enhance individual lines of code. The following instruction that determines if the input field we are about to

display will fit on the current line is almost unintelligible when written without any spacing to help differentiate its various components:

```
if(Field.Descriptor.all'length+Field.x+Screen_Delimiter'length+
short_integer(Field.Input_length)+String_at_End_of_Input'length
+Length_of_Multi_Line_Sign)>Text_Screen_Width
```

Adding space to separate various parts of this long instruction makes it easier to understand:

```
if (Field.Descriptor.all'length          +
    Field.x                              +
    Screen_Delimiter'length              +
    short_integer(Field.Input_length)    +
    String_at_End_of_Input'length        +
    Length_of_Multi_Line_Sign)                 >   Text_Screen_Width
```

3.4.7. DOCUMENT THINGS EXPECTED TO CHANGE

Document areas of code that are expected to change. One way to accomplish this is to thoroughly document data structures, especially complex records and arrays. In the SARAH project, displaying a message mask on the screen and capturing input data for specific fields is controlled by an internal table. After a brief explanation, comments defining the use of each record component, made it possible for a programmer new to the message mask table structure to add a table for a new message mask.

A very visual way that areas of code expected to change are documented is use of a procedure called Stub. Stub opens a transient window and displays a message that a certain feature of the system is not available. The hooks for features that will be implemented at a later date can be put into the code now. Then, Stub can be called if the missing feature is requested.

4. SUMMARY AND RECOMMENDATIONS

4.1. SUMMARY

Requirements changes, hardware advancements, and programming errors lend truth to the statement by Grady Booch that "Large software systems don't die; they simply get modified."[12] We can drastically reduce the overall costs of system software if we keep this in mind and design software systems with modifiability as one of our engineering goals.

The SARAH project team has implemented specific techniques to enhance the modifiability of SARAH software. The primary tool used to construct modifiable software is a sound design methodology. SARAH uses a combination of proven design methodologies as described in "An Architectural Approach to Developing Ada Software Systems", another paper in this series.

In addition to a sound design foundation, the SARAH project team has kept modifiability in mind as code is written and documented. Modularizing software components, especially machine dependent code and common tools packages, limits the scope of modifications. Several documentation conventions were implemented that have made the code produced by the SARAH team easier to read, easier to understand and, thus, easier to modify.

This paper is not intended to be an exhaustive list of all factors that promote software modifiability. It presents specific examples of the type of issues that should be considered with the hope that other software development projects will be able to adapt them to increase the modifiability of their software efforts.

4.2. RECOMMENDATIONS

Recommendations are:

- -- Use sound software engineering techniques to plan for software modifiability
- -- Use modularization techniques to reduce the overall system complexity and to isolate the effects of any modifications made
- -- Use comments to thoroughly document the structure and code of your system
- -- Create and follow coding/documentation standards

A. SARAH PROJECT -- CODING GUIDELINES FOR COMMENTS

In addition to the Intellimac Ada coding standards, we have considered the following additional guidelines that apply to program comments:

-- Each compile unit/major package should have a box of asterisks at the top

```
   --  within this box, show:
       --  the unit name
       --  its full Ada name
       --  its MS-DOS filename
       --  designer, programmer, etc. if appropriate
       --  a narrative description of what it is/does
```

-- If you wish to have "major sections" within the spec or program, you may mark these withe the name/description in a small box. It should have dotted lines all the way across the page. Example:

```
                      ------------------------
------------------->  CALCULATE CRC SECTION  <-------------------
                      ------------------------
```

To help locate functions and procedures, show their name in the same type box as above. However, start the middle line of dashes in approximately column 25. Example:

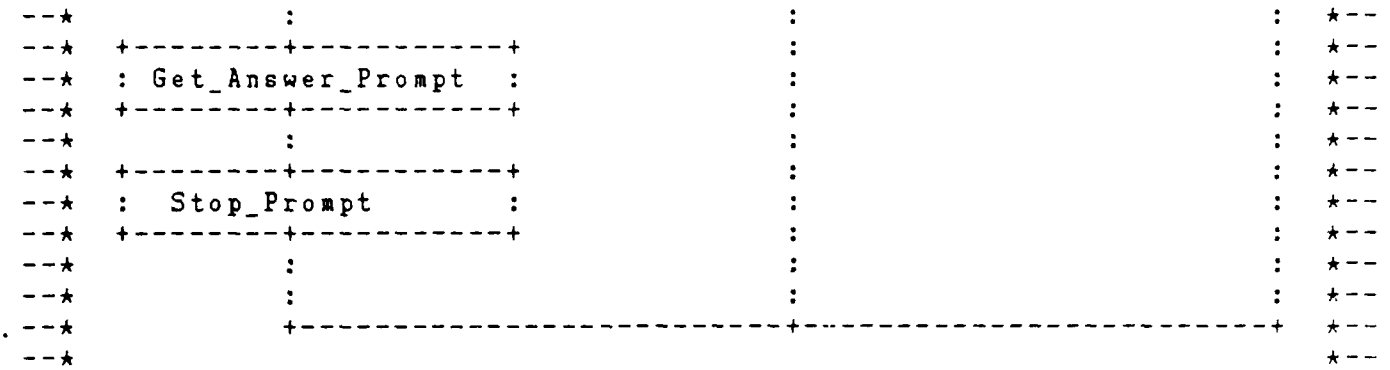
```
                                  --------------
                        ------------>  DISPLAY_MENU  <-------------
                                  --------------
```

-- All comments (except for those in the box of stars at the top) will start in approximately column 25. This will help keep the comments out of the way when you just want to "read the code". You can the look over to the right side of the page if you want to "read the comments". A single line of dashes should surround (top and bottom) the comments. Example:

```
                        -----------------------------------------
                        --  This   procedure  allows  writing  a
                        --  single character to the menu bar area.
                        --  The  'x'  parameter  is  the  character
                        --  location on the menu bar where the
                        --  character is to be written.
                        -----------------------------------------
```

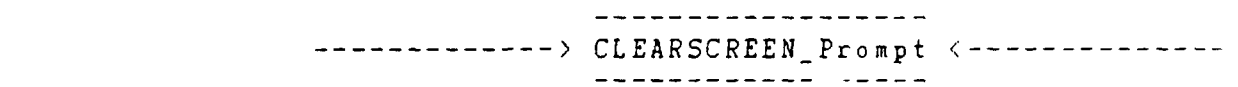
The following pages show these guidelines implemented for the PULLDOWN_MGR.

```
--*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*--
--*                        vdt_mgr\sarah\prom_mgr\prompmgr.ads                *--
--*                                                                           *--
--*                                                                           *--
--*                     PACKAGE SPEC Prompt_Window_Mgr                        *--
--*                             LLCSC 5.1.6                                   *--
--*                                                                           *--
--*                                                                           *--
--*      Design:  Oscar Staudt              Code:  Oscar Staudt               *--
--*               Susan Fannin                     Susan Fannin               *--
--*                                                                           *--
--*    This package contains the basic tools needed to control the            *--
--*    Prompt Window.  Users of this package may write short messages         *--
--*    or receive short replies from this window.                             *--
--*                                                                           *--
--*    NOTE:If control of the video attributes is needed, then                *--
--*         it is necessary to "with" the package SARAH_VDT_Constants         *--
--*                                                                           *--
--*                                                                           *--
--*            +-----------------------------------------------------+        *--
--*            :                  Prompt_Window_Mgr                  :        *--
--*            +----------------------------+------------------------+        *--
--*            :                            :                        :        *--
--*            :                            :                        :        *--
--*  +---------+------------+               :                        :        *--
--*  :  Initialize_Prompt   :               :                        :        *--
--*  +---------+------------+               :                        :        *--
--*            :                            :                        :        *--
--*  +---------+--------------+             :                        :        *--
--*  :  ClearScreen_Prompt    :             :                        :        *--
--*  +---------+--------------+             :                        :        *--
--*            :                            :                        :        *--
--*  +---------+------------+               :                        :        *--
--*  :  Put_CharX_Prompt    :               :                        :        *--
--*  +---------+------------+               :                        :        *--
--*            :                            :                        :        *--
--*  +---------+------------+               :                        :        *--
--*  :  Put_StrX_Prompt     :               :                        :        *--
--*  +---------+------------+               :                        :        *--
--*            :                            :                        :        *--
--*  +---------+--------------+             :                        :        *--
--*  :Set_Fore_Attrib_Prompt:               :                        :        *--
--*  +---------+--------------+             :                        :        *--
--*            :                            :                        :        *--
--*  +---------+--------------+             :                        :        *--
--*  :Set_Back_Attrib_Prompt:               :                        :        *--
--*  +---------+--------------+             :                        :        *--
--*            :                            :                        :        *--
--*  +---------+--------------+             :                        :        *--
--*  :Set_Blnk_Attrib_Prompt:               :                        :        *--
--*  +---------+--------------+             :                        :        *--
--*            :                            :                        :        *--
--*  +---------+-------------+              :                        :        *--
--*  :Set_Int_Attrib_Prompt:                :                        :        *--
--*  +---------+-------------+              :                        :        *--
```




```
With Key_Mgr;                Use Key_Mgr;
With Help_Window_Mgr;        Use Help_Window_Mgr;
With VDT_Types;              Use VDT_Types;
With Virtual_VDT_Tools;      Use Virtual_VDT_Tools;
With SARAH_VDT_Constants;    Use SARAH_VDT_Constants;

Package Prompt_Window_Mgr is

                             ------------------
   ----------------------->  Constants & Types <-----------------------
                             ------------------

   Subtype Prompt_X_Coord_Type is X_Coord_Type range 1..Prompt_Max_Cols;
   Subtype Prompt_Y_Coord_Type is Y_Coord_Type range 1..Prompt_Max_Rows;

   Prompt_Screen_Width: constant:= Prompt_Max_Cols;
   Prompt_Screen_Length: constant:= Prompt_Max_Rows;


                             ----------------
   ----------------------->    PROCEDURES     <-------------------------
                             ----------------


                                       --------------------
                           ----------->  Initialize_Prompt  <----------
                                       --------------------

   Procedure Initialize_Prompt;

                           ------------------------------------------------
                           --              unit 5.1.6.1
                           --
                           -- DESCRIPTION:
                           -- This procedure is called by the Initialize
                           -- procedure in the parent package (SARAH_VDT_
                           -- Tools).  It does all the internal processing
                           -- necessary to initialize the Prompt Window.
                           -- The functions performed are:
                           --    1. Get a window_ID for use throughout
                           --       the life of the Prompt Window
```

```
                    --    2. Set the default write attribute
                    --    3. Clear the Prompt Window
                    --
                    --
                    -- NOTE:
                    -- These functions are internal considerations
                    -- for Prompt Window Mgr., and need not concern the
                    -- user of this package.  In a testing
                    -- environment, this procedure will need to be
                    -- called by any routine doing testing without
                    -- the entire system environment.
                    --
                    ------------------------------------------------
```




```
Procedure Clearscreen_Prompt;

                    ------------------------------------------------
                    --                unit 5.1.6.2
                    --
                    -- DESCRIPTION:
                    -- This procedure will clear the "screen".
                    -- relative to the Prompt Window.
                    -- By definition, clearing the screen means
                    -- writing blank characters to all positions
                    -- within the Prompt Window.
                    --
                    --
                    -- NOTE:
                    -- The Default Write Attribute is used as the
                    -- video attribute when the character is written.
                    -- You must set it to the colors you want before
                    -- calling this procedure.
                    --
                    -- NOTE:  Since CLEARSCREEN_PROMPT is not visible
                    -- in the spec, it must be declared before it is
                    -- used by any routine in the body.
                    -- (ie. Initialize_Prompt)
                    ------------------------------------------------
```

```
                                    ------------------
                        ------------>  Put_CharX_Prompt  <--------------
                                    ------------------
Procedure Put_CharX_Prompt(X : IN      Prompt_X_Coord_Type;
                           Ch: IN      Character);

                        ------------------------------------------------
                        --                 unit 5.1.6.3
                        --
                        -- DESCRIPTION:
                        -- This procedure will write a single character
                        -- at the screen coordinate X (column) relative
                        -- to the the Prompt Window.
                        --
                        -- INPUT PARAMETERS:
                        -- X       = the horizontal coordinate
                        --           (x axis or column) where the character
                        --           is to be placed within the Main
                        --           Menu Bar
                        -- Ch      = the ASCII character to be placed at
                        --           the X coordinate of the Prompt Window.
                        --
                        --
                        -- NOTE:
                        -- The Prompt Window has only one line,
                        -- therefore it is not necessary to input a Y
                        -- coordinate to this procedure.
                        --
                        -- NOTE:
                        -- The first position in the X coordinate
                        -- system is 1 (not 0).  Location 1 is the first
                        -- column of the Prompt Window.
                        --
                        -- NOTE:
                        -- The Default Write Attribute is used as the
                        -- video attribute when the character is written.
                        -- You must set it to the colors you want before
                        -- calling this procedure.
                        --
                        -- NOTE:
                        --  This procedure is totally independent of the
                        --  positioning of the cursor.  If you wish the
                        --  cursor to move to a different position, that
                        --  must be done with the CursorX_Menu procedure.
                        --
                        ------------------------------------------------
```

B. REFERENCES

[1] "SARAH Operational Concept Document", Command and Control Systems Office, US Air Force, 5 September 1986.

[2] Merriam-Webster Dictionary, G. & C. Merriam Co., division of Simon & Schuster, New York, New York, 1974, pp 451.

[3] "An Architectural Approach to Developing Ada Software Systems", Headquarters United States Air Force, Information Systems Technology and Security Division, Washington D.C., 21 May 1986.

[4] Booch G., "Software Engineering with Ada", Benjamin/Cummings Publishing Company, Inc., Menlo Park, California, 1983.

[5] Gardner M, et all, "Ada Programming Style", INTELLIMAC, Inc., Rockville, Maryland, 1983.

[6] "Defense System Software Development", Mil-STD 2167, Appendix D (draft), Department of Defense, Washington D.C.

[7] E. Yourdon and L.L. Constantine, Structured Design, Englewood Cliffs, NJ: Prentice-Hall, 1979.

[8] Booch G., "Object Oriented Development", IEEE Transactions on Software Engineering, Vol. SE-12 No. 2, February 1986.

[9] "SARAH Operational Concept Document", Command and Control Systems Office, US Air Force, 5 September 1986.

[10] U.S. Department of Defense, "Reference Manual for the Ada Programming Language", ANSI/MIL-STD 1815A, Jan 1983.

[11] Ausnit C., Cohen N., Goodenough J., and Eanes R., "Ada in Practice", Springer-Verlag, West Hanover, Massachusetts, 1985.

[12] Booch G., "Software Engineering with Ada", Benjamin/Cummings Publishing Company, Inc., Menlo Park, California, 1983, page 25.

[13] Boehm, B. W., "Software and Its Impact: A Quantitative Assessment", Datamation, May 1973.

[14] "Ada training for Development Teams", Headquarters United States Air Force, Information Systems Technology and Security Division, Washington D.C., 13 March 1986.